

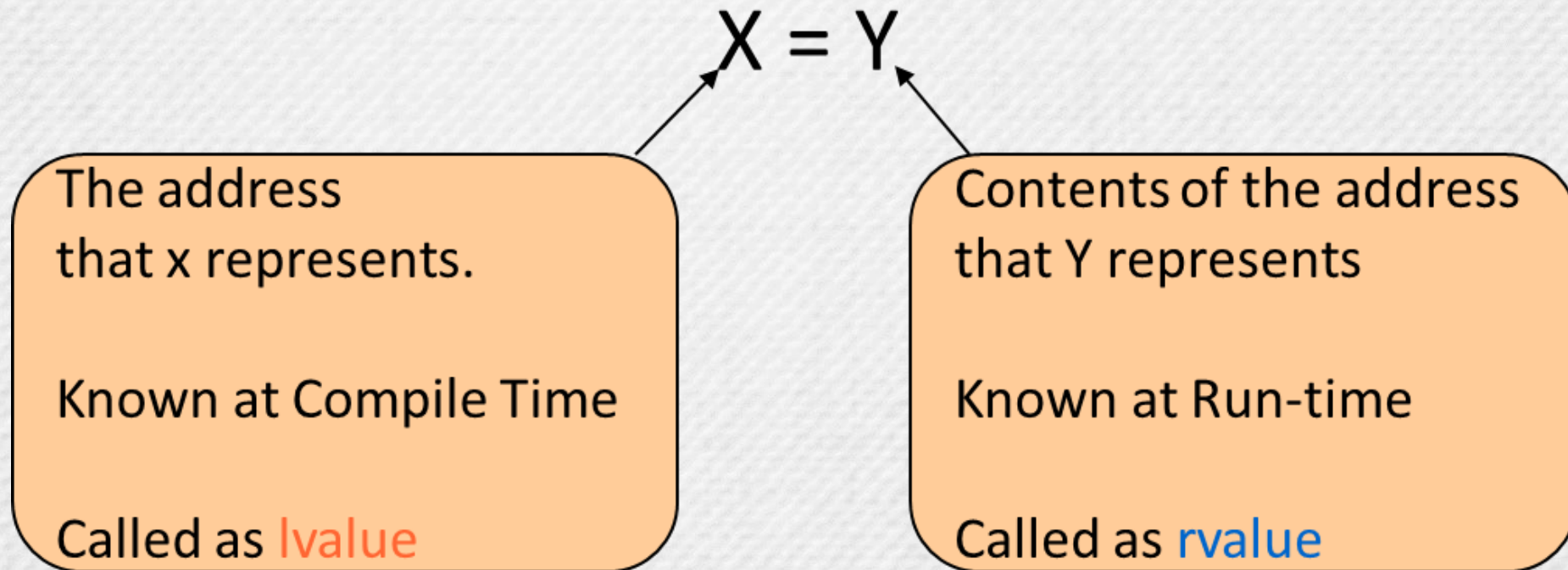
Pointers in C Programming

by
Ravi Kishore K

- Pointers are also variables (storing addresses), so they have a memory location, so they also have an address
- Pointer is a variable which holds the **address** of **similar** type of data.
- A Pointer, like an integer, holds a number.
 - Interpreted as the address of another object.
- Must be defined with its type
 - Pointer to char, pointer to int, etc...
- Size of any type of pointer = size of int = number of bits of a processor
 - Pointer holds an address, which is always an integer

- Pointer to pointer – stores the address of a pointer variable
- A pointer can point to another pointer of any type.
 - But use pointer to point to same type of pointer
- Can nest to 12 levels of indirection
 - Double pointers are the practical limit.

LVALUE & RVALUE



POINTER ARITHMETIC - EXAMPLE

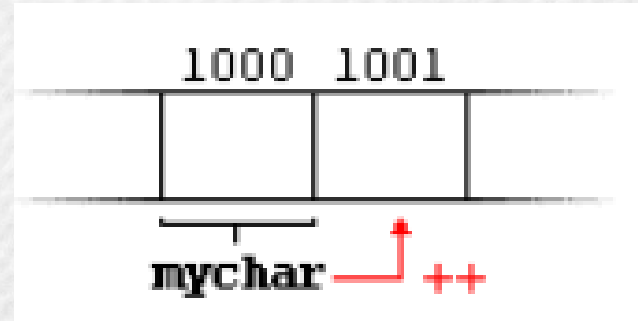
Data type	Size
char	1
short	2
long	4

```
char *mychar;
```

```
short *myshort;
```

```
long *mylong;
```

```
mychar++;
```



POINTER ARITHMETIC - EXAMPLE

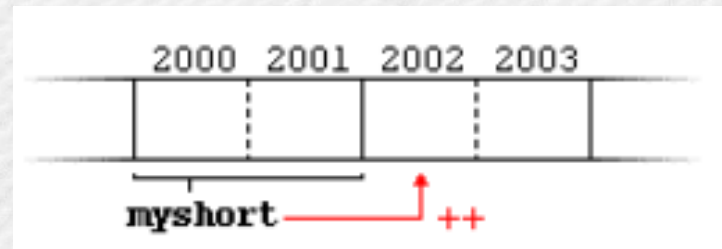
Data type	Size
char	1
short	2
long	4

```
myshort++;
```

```
char *mychar;
```

```
short *myshort;
```

```
long *mylong;
```



POINTER ARITHMETIC - EXAMPLE

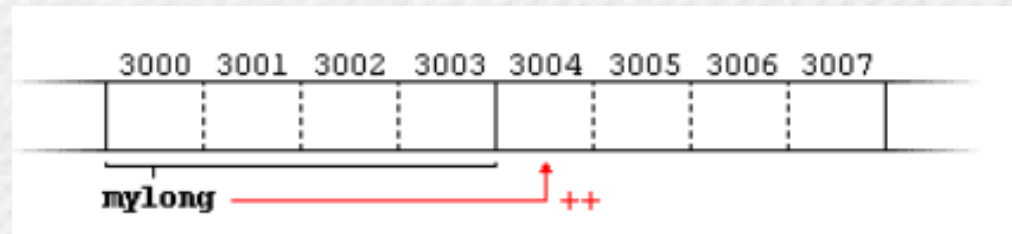
Data type	Size
char	1
short	2
long	4

```
mylong++;
```

```
char *mychar;
```

```
short *myshort;
```

```
long *mylong;
```



- You can add/subtract an scalar value (integer) to/from a pointer.
- The pointer advances/ retreats that **number of elements**
 - **Not the number of bytes.**
- Subtracting 2 pointers yields to number of elements between them.
- Multiplication & Division operations cannot be performed on pointers

POINTER ARITHMETIC CONT..!!

`Ptr = &array[0];` //Point to the 1st element of array

`Val = *ptr++;` //Get element #0, ptr points to #1

`Val = *++ptr;` //Get element #2, ptr points to #2

`Val = ++*ptr;` //Increment #2, return its value

- Pointer to void (**void ***)
- When other pointer types are assigned to generic pointer, conversions are applied automatically.
- Any pointer can be casted to void* and back again without loss of information.
- Can assign any pointer to or from a void*.
 - Need to cast back to use the item
 - Can not de-reference a void*
- Useful For:
 - Treating any object as a sequence of bytes.
 - Assignment and initialization
 - Comparision

```
int main()
{
    int a = 10;
    float x = 5.7;
    void *p;
    p = &a;
    printf("value of a = %d\n", *((int *)p));
    p = &x;
    printf("value of x = %f\n", *((float *)p));
}
```

- A Special Value- The address 0
- Doesn't point anywhere
- It can be used to compare to other pointers.
 - Eg: if malloc() fails returns NULL pointer.
- Can not dereference NULL.

- Gives the size of a variable or type in bytes.
- A compile-time operator
- Array size idiom
 - `int n = sizeof a / sizeof a[0];`
- Must use parentheses with types.
 - `float *p = (float *)malloc(sizeof(float));`

- Prevents Changing a pointer or its CONTENTS.
- Const before the asterisk means *content* is constant.
- Const after the asterisk means that the *pointer* is constant.

```
const char* p1;      /* *p1 = 'c' illegal; ++p1 OK */
char* const p2;      /* *p2 = 'c' OK; ++p2 illegal */
const char* const p3; /* no changes at all allowed */
```

DYNAMIC MEMORY ALLOCATION

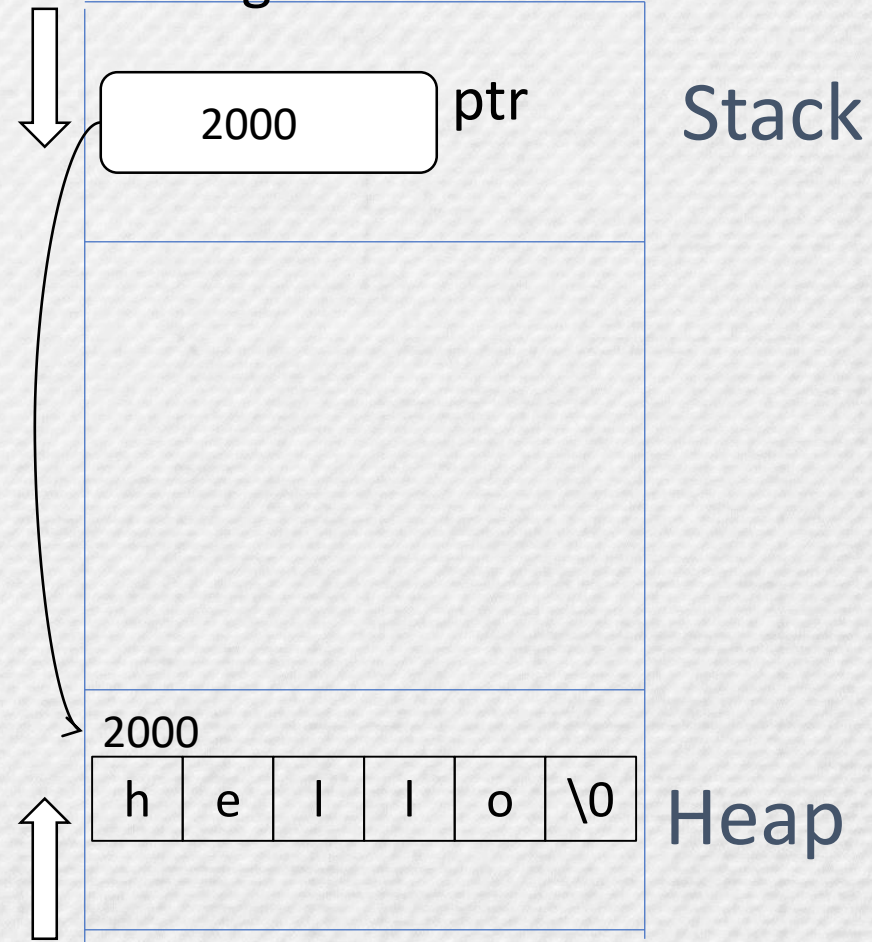
- Sometimes
 - Amount of data cannot be predicted beforehand
 - Number of data items keeps changing during program execution
- Example: Maintaining Student Records
 - Number of students of each organization is different
 - If arrays are considered then number of students(rows) and length of student name(columns) should be known beforehand
 - Which wastes lots of memory if the input is less than the allocated memory
 - Need to develop a software for addressing the needs of any organization

- Dynamic memory allocation
 - Know how much memory is needed after the program is run
 - Example: ask the user to enter from keyboard
 - Dynamically allocate only the amount of memory needed
- C provides functions to dynamically allocate memory
 - malloc, calloc, realloc

DYNAMIC MEMORY

```
char *ptr;  
ptr =(char *) malloc(6*sizeof(char));
```

Storage for Local Variables
& Program Execution



Storage for Dynamic Memory

```
void *malloc(size_t size);
```

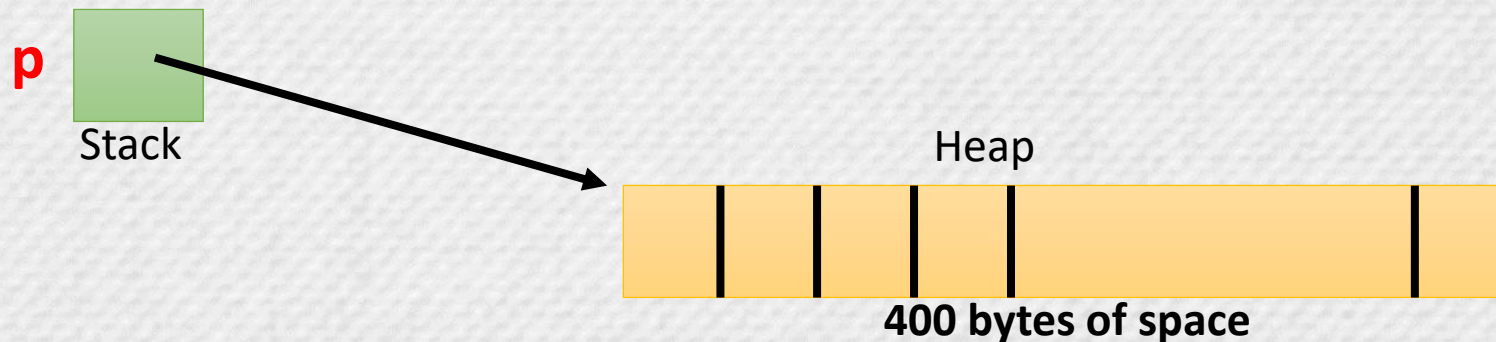
- Allocate a block of `size` bytes
 - On Success: returns a pointer to the allocated memory
 - On Failure: returns `NULL` pointer
 - Doesn't initialize the allocated memory contents

```
void *calloc(size_t num_elements, size_t element_size);
```

- Allocate a block of `num_elements * element_size` bytes
 - On Success: returns a pointer to the allocated memory
 - On Failure: returns `NULL` pointer
 - initialize every byte to zero

```
p = (int *) malloc(100 * sizeof(int));
```

- A memory space equivalent to **100 times the size of an int** bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer **p** of type **int**



```
void free(void *pointer);
```

- put the region back in the heap of unallocated memory
- An allocated block can be returned to the system for future use by using the **free** function
- **No Size needs** to be mentioned for the allocated block, the system remembers it for each pointer returned
- Note: easy to forget to free the memory when no longer needed...
 - This leads “memory leak” problem
 - Difficult to trace – the program will run fine for some time, and suddenly there is no more memory!

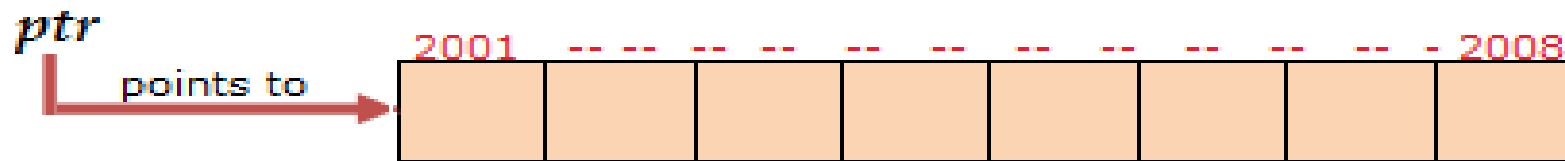
```
void *realloc(void *ptr, size_t new_size);
```

- Given a previously allocated block starting at `ptr`,
 - change the block size to `new_size`,
 - return pointer to resized block
- If block size is increased, contents of old block may be copied to a completely different region
- In this case, the pointer returned will be different from the `ptr` argument, and `ptr` will no longer point to a valid memory region

REALLOC TO INCREASE SIZE

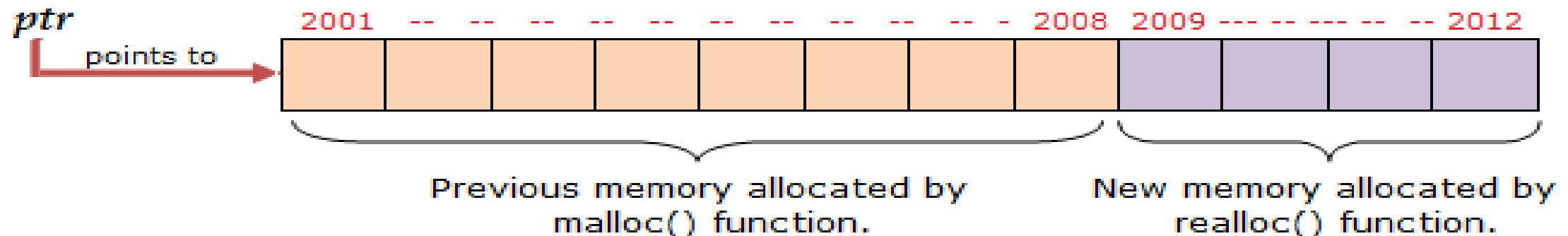
```
int *ptr;  
ptr = (int *)malloc(2*sizeof(int));
```

8 bytes



If we want to increase the size by 4 more bytes,

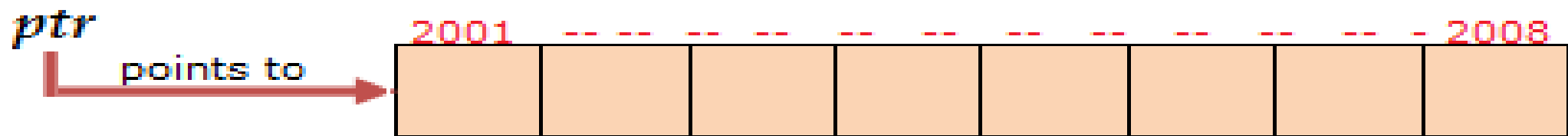
```
ptr = (int *)realloc(ptr,12);
```



REALLOC TO REDUCE SIZE

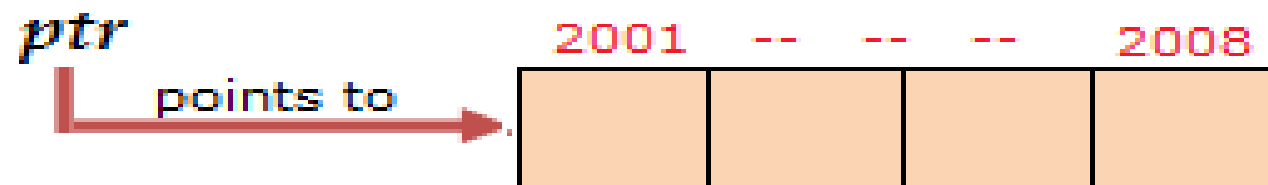
```
int *ptr;  
ptr = (int *)malloc(2*sizeof(int));
```

8 bytes



If we want to decrease the size by 4 bytes,

```
ptr = (int *)realloc(ptr,4);
```



- ✓ If `ptr` is `NULL`, then call is equivalent to `malloc(size)`;
- ✓ If `size` is zero, then the call is equivalent to `free(ptr)`;
- ✓ If `there is not enough memory` to expand the block to given size, `original block is unchanged and NULL is returned`. Original block is not freed or moved.

- Call to `malloc` might fail to allocate memory, if there's not enough available

```
void *new_mem = NULL;
new_mem = (type *)malloc(size);
if (new_mem == NULL) {
    printf("Give some message to user");
    return -1; // terminates the application
}
printf("new memory successfully created");
```

- **Memory Leak**

Memory is existing but pointer is not there

```
m = malloc(5);
```

```
m = NULL;
```

Allocated 5 bytes are still exists in memory but we cant access them because we lost the reference to that created memory

- **Dangling Reference**

Memory is not there but pointer is existing

```
p = malloc(5);
```

```
Strcpy(p, "bye");
```

```
q = p;
```

```
free(p);
```

Printf("%s\n", q); -> Data is not present but q is still a valid reference. This is called as Dangling Reference

THANK YOU